Hybrid Genetic Search for the Vehicle Routing Problem with Time Windows: a High-Performance Implementation

Wouter Kool
Joep Olde Juninck
Ernst Roos
Kamiel Cornelissen
Pim Agterberg
Jelke van Hoorn
Thomas Visser
ORTEC

Email of Corresponding Author: wouter.kool@ortec.com

Abstract: This paper describes a high-performance implementation of Hybrid Genetic Search (HGS) for the Vehicle Routing Problem with Time Windows (VRPTW) [11]. We added time window support to the state-of-the-art open-source implementation of HGS for the Capacitated Vehicle Routing Problem (HGS-CVRP) [10], and included additional construction heuristics, a Selective Route Exchange (SREX) [6] crossover and an intensified local search procedure inspired by the `SWAP*` neighborhood [11]. The code has been optimized and we used different schedules for growing the size of neighborhood and population based on instance characteristics. For the VRPTW with distance-only objective (not minimizing vehicles) we found several improvements of best known solutions (BKS) for Gehring & Homberger [3] benchmark instances. The solver ranked 1st in Phase 1 of the VRPTW track of the 12th DIMACS implementation challenge.

Team name: Wouter & Co
Solver name: Router
VRP tracks: VRPTW
Code source: [to be released]
Date: 1 Feb 2022

1 Introduction

This paper describes a high-performance implementation of Hybrid Genetic Search (HGS) for the Vehicle Routing Problem with Time Windows (VRPTW) [11], based on a state-of-the-art open-source implementation of HGS for the Capacitated Vehicle Routing Problem (HGS-CVRP) [10], which we adapted to support time windows by adding a time-warp functionality with penalties [11]. We optimized the performance and added extra construction heuristics, a new method for generating offspring (combining solutions in the genetic algorithm) and a local search intensification procedure inspired by the `SWAP*` operator [10]. Tuning of the parameters resulted in a schedule that gradually grows the size of the neighborhood and genetic population. The solver ranked 1st in Phase 1 of the VRPTW track of the 12th DIMACS implementation challenge[1].

1.1 The 12th DIMACS implementation challenge: VRPTW track

The VRPTW variant considered in the DIMACS challenge has two important properties: the objective is to minimize *distance only* and the distance is measured as the *Euclidean distance truncated to one decimal*. The number of available vehicles is given per instance but provided no practical limitation for the 56 Solomon [8] and 300 Gehring & Homberger [3] instances considered. For these instances, the DIMACS organization provided a set of *reference solutions* that were used to compute the *primal integral* (PI) [2] for evaluating solvers submitted to the challenge. We call these reference solutions, rather than best known solutions (BKS), as feasible solutions with shorter total distance (with 1-decimal precision) were already registered by SINTEF[2] (even though these used double precision and the number of vehicles as primary objective). For the distance-only objective, we found new BKS, improving the reference and SINTEF solutions for many instances.

2 Hybrid Genetic Search for the VRPTW

The basis of our algorithm is HGS-CVRP [10][3]: a state-of-the-art open-source genetic algorithm. It maintains a pool (or *population*) with feasible and a pool with infeasible solutions. Initially, 100 random solutions are created, by using the `SPLIT` algorithm[1, 9] on a random ordering of the customers, and inserted in the right pool based on feasibility. In every iteration, two parents (feasible or infeasible) are selected from the pools using a *binary tournament*, which are combined using an *ordered crossover* [7] to create a new *offspring solution*, which is then improved using a local search. Violations of capacity constraints incur a penalty that is automatically adjusted such that $\pm 20\%$ of the local searches results in a feasible solution. Once a pool size reaches $\mu + \lambda$ solutions, the solutions with lowest *fitness* (taking into account quality and diversity) are removed one by one (updating the fitness after each removal) until μ solutions remain ($\mu = 25, \lambda = 40$).

[1] `http://dimacs.rutgers.edu/programs/challenge/vrp/vrptw/`

[2] `http://www.sintef.no/vrptw`

[3] `https://github.com/vidalt/HGS-CVRP`

We adapted the HGS-CVRP code to support time windows and added construction heuristics, an improved crossover procedure and an intensified local search. We added caching mechanisms and pre-checks for local search operators, and made performance optimizations: as we required 1-decimal precision, we multiplied distances (and time windows) by 10 and used integer arithmetic, giving a large speed-up. Additionally we implemented the distance matrix as a flat array, which is more efficient than a `c++` vector of vectors. Finally, we removed unnecessary route-duration checks.

2.1 Supporting time windows

To support time windows, we follow the original HGS paper for VRPTW [11]. With time windows, a vehicle must arrive at a customer between an earliest and latest arrival time, after which it requires a certain service time before it can continue to the next customer. Following [11], we implement the *time-warp* principle [5], which lets the vehicle 'travel back in time' to the latest arrival time if it arrives too late at a customer. The total time-warp is multiplied by a penalty weight and added to the objective. The time-warp penalty weight is initialized at 1 and adjusted every 100 iterations, similarly to the capacity penalty: it is increased by 20% if less than 15% of solutions resulting from the local search is feasible (w.r.t. time windows) and decreased by 15% if more than 25% is feasible. Additionally, we added a *penalty booster* that increases the penalty by 100% if no feasible solution (w.r.t. time windows) has been found yet: otherwise it can take long before a feasible solution is found. We found this to work well, as opposed to using a larger initial penalty from the start, which lets the algorithm converge too quickly to suboptimal solutions, hurting final performance.

The total time-warp can be computed efficiently by tracking certain statistics for subsequences of nodes within a route [11], which we refer to as TW-data (time window data). Different from [11], for each customer in a route, we precompute (cache) the *prefix* TW-data (corresponding to the start of the route: the sequence from the depot up to and including that customer) and the *postfix* TW-data (corresponding to the end of the route: starting at the customer and ending at the depot). This enables the computation of the total time-warp for a route after insertion or removal of a customer (but not both) in $O(1)$. For simultaneous insertion and removal at different positions in a route (e.g. for `RELOCATE` or `SWAP*`, see Section 2.4), we need to compute the TW-data for the route segment between the two positions involved: for this we use a two level hierarchical datastructure [4] that precomputes TW-data for smaller route segments between seed customers (one every 4 customers), which can be combined to compute TW-data for longer route segments.

2.2 Construction heuristics

The main goal of the initial solutions in the pools is to provide diversity for the genetic algorithm to succeed. For this reason, [10] uses random initial solutions, which then get improved through the local search. As we desire to minimize the primal integral, we add three additional construction heuristics, *nearest*, *farthest* and *sweep*, each of which constructs 5% of the 100 initial solutions, with the goal of (besides diversity) finding feasible solutions of reasonable quality quickly.

With *nearest* and *farthest* (inspired by VROOM[4]), we construct routes one at a time, starting a route by inserting the unassigned customer that is nearest to/farthest from the depot, respectively. Then, we repeatedly find the combination of unassigned customer and insertion position that causes the least detour distance and does not cause any time window or vehicle capacity violations, and insert it. If no customers can be inserted without causing violations, we start the next route. These heuristics are deterministic, so to construct more than one solution, we construct additional (almost feasible) solutions by allowing each route to use a small time-warp (with a random tolerance between 0 and 100) and violate the capacity constraint (with a tolerance between 0 and 50).

With *sweep*, we sort the customers according to angle relative to the depot. We construct the routes one at a time. We keep assigning customers to the route from the start of the sorted list of unassigned customers until capacity would be violated. To determine the order of the customers within the route, first all customers with short time windows (at most half of the planning horizon) are sorted increasingly by latest arrival time. After that, the remaining customers are inserted (in arbitrary order) in the partially constructed route at the position that causes the least detour distance. Note that this heuristic may construct routes that violate time windows. To construct additional solutions, we use the heuristic with a capacity that is randomly reduced by 0% to 40%.

2.3 Offspring generation

HGS uses an *ordered crossover* (OX) [7] to create new offspring from two parents that are selected using *binary tournaments*: each parent is the best of two random solutions in the pool, according to the *fitness* (considering quality and diversity) [10]. The ordered crossover randomly selects a subsequence of nodes from the first parent (disregarding any depot visits) as a basis, and appends the missing nodes in the order they appear in the second parent. Depot visits are re-inserted in the solution using the `SPLIT` [1, 9] algorithm, which disregards time windows, but we rely on local search to resolve any time window violations. Compared to HGS-CVRP [10], we change the maximum number of routes for the `SPLIT` algorithm: we use all available vehicles, rather than the number of routes of the first parent, as we do not need to minimize the number of vehicles used.

We improve the offspring generation by adding a second crossover operator: *Selective Route Exchange* (SREX) [6]. SREX combines entire routes from the two parents to generate offspring solutions. Specifically, a set of routes is randomly selected from one parent and replaced by a set of similar routes from the other parent. Nodes that appear in two routes are all removed from routes originating from a single parent. Nodes that are missing are inserted into a route based on detour distance. Two different offspring solutions are created dependent on from which parents' routes the duplicate nodes are removed and we continue with the best solution (in terms of penalized cost). Since SREX preserves depot visits, it does not use the `SPLIT` algorithm and is thus more suitable for time windows. Finally, we generate offspring with both OX and SREX and continue to the local search phase with the best (in terms of penalized cost) offspring out of these two candidates.

[4] `http://vroom-project.org/`

2.4 Local search & SWAP* neighborhood

The local search consists of the SWAP, RELOCATE, 2-OPT and 2-OPT* moves described in [11, 10]. Each move can be described by a pair of nodes (u, v). In [11], only moves defined by promising arcs (u, v) are considered, which is measured by an asymmetric measure $\gamma(u, v)$, taking into account both spatial and time proximity (Eq. (4) in [11]). Since most moves do not actually result in the arc (u, v) being part of the resulting solution (e.g. when swapping u and v), we use the symmetric measure $\hat{\gamma}(u, v) = \min\{\gamma(u, v), \gamma(v, u)\}$ instead. For each node, we only consider moves involving the Γ 'nearest' neighbors in terms of $\hat{\gamma}(u, v)$. We avoid redundantly checking symmetric operators for (u, v) and (v, u), and include pre-checks based on bounds to avoid expensive TW-data computations (see Section 2.1): if the routes involved in a move currently have 0 time-warp, then we first check only if the move will reduce the total distance. If this is not the case, the move can never be improving and we can discard the move, avoiding the expensive TW-data computation.

Intensification: SWAP* and RELOCATE* Every time a new local search is started (after constructing a solution or generating new offspring), with a certain *intensification probability* θ we run the search with a larger neighborhood inspired by the SWAP* neighborhood [10]: in this case, if none of the basic operators yield an improvement, we iterate over all pairs of routes with overlapping circle sectors (see [10]), where we enforce a minimum circle sector size of 15° to guarantee overlap for heavily clustered routes. For each pair of routes, we try sequentially the RELOCATE* operator, the distance-based SWAP* operator and the TW-data-based SWAP* operator. If any of these yield an improvement, we continue with the basic operators and repeat.

The RELOCATE* operator tries to move a node from one route to the best position in the other route. Worst case, this operator considers all RELOCATE moves thus the operator is $O(n^2)$. The SWAP* operator aims to exchange two nodes between the two routes, inserting them in the best position in the other route. For the CVRP, the SWAP* operator is exact in the sense that it always finds the best move, which is achieved efficiently (in overall $O(n^2)$ computation) by precomputing the top 3 insertion positions (based on distance) for each node in the other route. After removing another node, the best insertion position is still one of the top 3 positions, or the position of the removed node, so we can safely ignore all other positions. With time windows, this property is not guaranteed, but we can still consider only the top 3 insertion positions as a heuristic. The distance-based SWAP* operator assumes that good moves should not cause large detours, and simply finds the best SWAP move based on distance (i.e. ignoring time windows), and then tries this move. The TW-data-based SWAP* simply assumes that the effect on the total time-warp of removing a customer from a route, and adding another customer, is independent, such that the change in time-warp penalties from both operations can be added. In this case, we can efficiently precompute the top 3 insertion locations for each node (including time-warp penalties) and compute the best SWAP move under this assumption. Like the distance-based SWAP*, the final cost (including actual time-warp penalties) is computed for the best move only. We found this move to be effective in practice.

2.5 Growing the population and neighborhood size

We found it beneficial to classify instances based on whether they have *short* or *long* routes (> 25 customers per route, estimated from average demand and vehicle capacity), and whether they have at least one customer with large time windows (more than 70% of the horizon). Based on these properties, we vary the intensification probability θ, and the schedule for growing the minimum population size μ and neighborhood size Γ. For instances with long routes, we use $\theta = 15\%$, $\mu = 25$, $\Gamma = 40$ and we grow both μ and Γ by 5 every 10000 iterations. For instances with short routes, we use $\theta = 100\%$ and we only grow the population size μ: we set $\Gamma = 40$ and grow $\mu = 25$ by 5 every 10000 iterations for instances *without* large time windows, but if an instance has (any) large time windows, we set $\Gamma = 20$ and we grow $\mu = 25$ by 5 only every 20000 iterations, to compensate for faster iterations. This way, relatively more time is spent on intensification, which is more effective with the flexibility from large time windows. After 10000 iterations without improvement, we restart the algorithm (generating a new population) without resetting parameters.

3 Results

We tested our solver using the DIMACS rules on 56 Solomon [8] and 5×60 Gehring & Homberger [3] instances. We multiplied the time limit of 30, 60 or 120 minutes for instances with ≤ 200, ≤ 800 and > 800 customers (excluding depot), respectively, with $\frac{2000}{2410}$ to standardize timings using the PASSMARK score of 2410 for our Intel i7-6850K CPU. Table 1 reports the final gap (percentage difference) to the DIMACS reference solution and *primal integral* (PI) [2] (average gap over the standardized computation time), for all groups of instances. Averaging over all groups, we obtain a gap of 0.004% and PI of 0.093%. In many cases, our algorithm has not converged and better results can be obtained with additional runs or runtime.

Dataset	C1	C2	R1	R2	RC1	RC2	Mean
Solomon	0,000%	0,000%	-0,003%	0,000%	0,000%	0,000%	0,000%
GH200	0,000%	0,004%	0,001%	0,009%	0,016%	0,026%	0,009%
GH400	0,000%	0,000%	-0,009%	0,028%	-0,030%	-0,050%	-0,010%
GH600	-0,014%	0,022%	0,047%	-0,022%	-0,012%	-0,123%	-0,017%
GH800	0,030%	-0,018%	0,147%	0,090%	0,112%	-0,222%	0,023%
GH1000	0,123%	-0,013%	0,174%	-0,090%	0,094%	-0,158%	0,022%
Mean	0,023%	-0,001%	0,060%	0,002%	0,030%	-0,088%	0,004%

(a) Gap to reference solution

Dataset	C1	C2	R1	R2	RC1	RC2	Mean
Solomon	0,000%	0,000%	-0,001%	0,002%	0,002%	0,001%	0,001%
GH200	0,001%	0,006%	0,014%	0,016%	0,025%	0,033%	0,016%
GH400	0,011%	0,014%	0,051%	0,065%	0,026%	-0,017%	0,025%
GH600	0,037%	0,060%	0,252%	0,128%	0,181%	0,010%	0,111%
GH800	0,082%	0,028%	0,424%	0,286%	0,312%	0,037%	0,195%
GH1000	0,207%	0,020%	0,479%	0,188%	0,319%	0,048%	0,210%
Mean	0,056%	0,021%	0,203%	0,114%	0,144%	0,019%	0,093%

(b) Primal Integral (PI)

Table 1: Results using a single run per instance according to the DIMACS challenge rules.

4 Conclusion

With the right components, parameters and implementation, Hybrid Genetic Search yields state-of-the-art results for the VRPTW, as was shown in the 12th DIMACS implementation challenge.

References

[1] John E Beasley. "Route first—cluster second methods for vehicle routing". In: *Omega* 11.4 (1983), pp. 403–408.

[2] Timo Berthold. "Measuring the impact of primal heuristics". In: *Operations Research Letters* 41.6 (2013), pp. 611–614.

[3] Hermann Gehring and Jörg Homberger. "A parallel hybrid evolutionary metaheuristic for the vehicle routing problem with time windows". In: *Proceedings of EUROGEN99*. Vol. 2. Citeseer. 1999, pp. 57–64.

[4] Stefan Irnich. "A unified modeling and solution framework for vehicle routing and local search-based metaheuristics". In: *INFORMS Journal on Computing* 20.2 (2008), pp. 270–287.

[5] Yuichi Nagata, Olli Bräysy, and Wout Dullaert. "A penalty-based edge assembly memetic algorithm for the vehicle routing problem with time windows". In: *Computers & operations research* 37.4 (2010), pp. 724–737.

[6] Yuichi Nagata and Shigenobu Kobayashi. "A memetic algorithm for the pickup and delivery problem with time windows using selective route exchange crossover". In: *International Conference on Parallel Problem Solving from Nature*. Springer. 2010, pp. 536–545.

[7] IM Oliver, DJd Smith, and John RC Holland. "Study of permutation crossover operators on the traveling salesman problem". In: *Genetic algorithms and their applications: proceedings of the second International Conference on Genetic Algorithms: July 28-31, 1987 at the Massachusetts Institute of Technology, Cambridge, MA*. Hillsdale, NJ: L. Erlhaum Associates, 1987. 1987.

[8] Marius M Solomon. "Algorithms for the vehicle routing and scheduling problems with time window constraints". In: *Operations research* 35.2 (1987), pp. 254–265.

[9] Thibaut Vidal. "Split algorithm in O (n) for the capacitated vehicle routing problem". In: *Computers & Operations Research* 69 (2016), pp. 40–47.

[10] Thibaut Vidal. "Hybrid genetic search for the cvrp: Open-source implementation and swap* neighborhood". In: *Computers & Operations Research* 140 (2022), p. 105643.

[11] Thibaut Vidal, Teodor Gabriel Crainic, Michel Gendreau, Nadia Lahrichi, and Walter Rei. "A hybrid genetic algorithm for multidepot and periodic vehicle routing problems". In: *Operations Research* 60.3 (2012), pp. 611–624.